CS 330 – Spring 2023 – Homework 05

Due: Friday 3/17 at 11:59 pm on Gradescope

Collaboration policy Collaboration on homework problems is permitted, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of your algorithm.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g. correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

- By: Gianna Sfrisi U77992006
- Collaborators: Albert Slepak U00956163

Problem 1. Unique shortest paths (10 points)

Although we typically speak of "the" shortest path between two nodes, a single graph could contain several minimum-length paths with the same endpoints. In this problem you will identify nodes that have multiple shortest paths from the source.

The input to your algorithm is the adjacency list of the weighted graph G and a source node s. We is such that it only has *positive* edge weights.

In this problem you will most likely apply a slight modification to Dijkstra's and/or call it as a subroutine. Note that in this case you don't have to write out the whole algorithm, rather specify what the modification is, or what the input and output is in case of the subroutine. You can also reference its correctness and running time when appropriate. Specifically, for correctness you can state that the shortest paths tree, i.e. parents list, returned by Dijkstra's indeed consists of shortest paths from s to each node. You do need to prove why the modifications solve the problem.

1. (not to be handed in) Draw some directed weighted graphs for yourself and identify the nodes that have multiple shortest paths from the source. Try to come up with ex
2. Design an algorithm that takes as input G and s and returns a hash table `Unique` that contains for every node v True/False whether there is a unique shortest path from s to v.

   Modification:
   In solving this problem, the original Dijkstra's algorithm from lecture must be modified to find if there are multiple shortest paths from s to v. In the original algorithm, there is a for loop and an if statement that tests if the current distance is less than the shortest distance found up until that point. While still inside the for loop, another clause was added to take into account situations where the same distance is found. This can be seen through the else if statement inside this particular for loop. The second modification made was in relation to the parents variable. In changing the parents variable into a list, the algorithm can keep track of all the parents of v that result in the shortest path. Additionally, in cases where a shorter distance is found, the parents list is reinitialized to empty. This prevents the parents resulting in a longer distance from being included in the final parents list. If the else if statement returns true, the second parent will be added to the list. The last modification was added at the end of the original algorithm and includes a for loop to cycle through the parents list of each node and check the length. If the length of the parents list is equal to 1, then only one shortest path was found, hence the path is unique. In this case the value in the Unique table is set to true. If the length of the parents list if greater than 1, that means multiple shortest paths were found and Unique is set to false for that node. The modifications made to the original algorithm are shown below.

   Run-time Analysis:
   The run-time of the original Dijkstra's algorithm as provided in lecture is $O(mlog(n))$. The modifications made to the original algorithm included adding the second if statement to the inner for loop, which would run in $O(1)$ time, and changing the parents variable to a list which does not affect the run-time. The last modification was including another for loop. The subroutine runs in time $O(n)$ as it will loop through each node one time, check the length, and change the Unique table value as necessary. The total run-time of the algorithm is $O(mlog(n)) + O(n)$ which can be simplified to $O(mlog(n))$.

```
// Modification to Dijkstra's Algorithm's inner for loop
for v in G[u] do
    if $\pi$[v] > d[u] + G[u][v] then
        /* reset the parents list for a node if a shorter path is found */
        parents[v] = [];

        $\pi$[v] = d[u] + G[u][v]
        parents[v].append(u);


    else if $\pi$[v] == d[u] + G[u][v] then
        /* In case the current distance is the same as the shortest distance, a.k.a another shortest
path was found */
        parents[v].append(u);

// After calling Dijsktra's, we run through all the nodes and check
// which have more than one parent for the shortest path.
for v in G do
    if parents[v].length > 1 then
        Unique[v] = false;
    else if parents[v].length == 1 then
        Unique[v] = true;
```

3. Now somebody has already run Dijkstra's algorithm for you on G and gave you the output of it (refer to slides 30 and 36 from March 2nd)

   ```
   d, parents = Dijkstra(G,s)
   ```

   Further, they told you that the node with id x is known to have multiple shortest paths from s to x.

   Design an algorithm that takes as input G, s, x, `d` and `parents` and returns the edges of *two* different shortest paths from s to x. (If there are more than two, then it doesn't matter which two are being returned.) For full credit your algorithm should run in time $O(n+m)$. *Hint: You may want to modify the backtracking algorithm that traces the paths given the parents.*

   Algorithm:

   ```
   /* First we call the modified Dijkstra's Algorithm from part 1B once on the weighted graph and get
   a list of parents for each node. */
   d, parents = DijkstraFrom1B(G, s);

   firstShortestPath = []; /* List to keep track of the first shortest path */
   secondShortestPath = []; /* List to keep track of the second shortest path */
   ```

```
/* This function will recursively backtrack up the parents from node v to source node s */ function
traceParents(s, v) {
        shortestPath = [];
         /* Sanity check the base case where the source node is reached */
         if (v == s)
                 return [shortestPath];

/* Checks which parent of v is a part of the shortest path */
        addParent = parents[v][0]

         /* Loops through each parent of v until the one that is a part of the shortest path is found */
         for i in parents[v] do
                 u = parents[v][i]; /* get the current parent */
                 if d[v] == d[u] + w(u, v)
                /* w(u,v) is the weight between v and its current parent being searched */
                /*if the distance matches the shortest distance*/
                         addParent = u;
                         break;
/* Once the parent that is included in the shortest path is found, it is added to the list and the recursive
function is called on that parent */
         shortestPath.append(addParent);
         traceParents(s, addParent);

/* End of helper function*/
firstShortestPath = traceParents(s, parents[v][0]);
secondShortestPath = traceParents(s, parents[v][1]);

/* We return the two paths*/
return [firstShortestPath, secondShortestPath]
```

Run-time Analysis:
The run-time of the modified algorithm from part 2 is $O(mlog(n))$. The recursive function cycles through the parent nodes and edges of each current node. As an edge is added to the edge list for the shortest path, the next node's parents are searched until the first one that matches the shortest distance is found. In the worst possible case every node and edge will be searched, hence the overall run-time for the recursive function will be $O(n + m)$. Therefore the total run-time between the modified Dijkstra's algorithm and the recursive function is $O(mlog(n)) + O(n + m)$ which simplifies to $O(n + m)$.

Proof of Accuracy:
In using the modified Dijkstra's algorithm from part 2, a list of parents of the shortest possible path is returned. As stated in the problem, there are multiple shortest paths from s to x, hence in calling the modified algorithm, a list of all the possible parents of x will be returned. The algorithm then uses recursive backtracking on the parent list to return two lists of edges that make up the shortest distance from s to x. Since it does not matter which edge list is returned, the algorithm automatically uses the first two parents in the parent list of x. The full process is called twice; once for the first parent in the parent list in x and a second for the second parent. To start, the backtracking algorithm checks the base case to see if the current node (which at the start is the parent of x) being checked is s; if the current node is equal to s, then the function returns the list of edges as the full path to make up the shortest distance as been found. In the case that the full path is not yet found, the algorithm

uses a for loop to cycle through the parents of the current node to find which parent makes up the shortest path found by Dijkstra's algorithm. Once that parent is found, it is added to the edge list, the recursive function is called on that parent node and the cycle continues until the current node matches the source node. Once the full process occurs twice, the algorithm returns two lists of edges that are the shortest paths from s to x.

Problem 2. Trip planning (10 points)

You are planning a vacation in a faraway country, and are looking at the train schedules in that country. The train schedule is made up of a set of n 4-tuples.

$$train(i) = (depcity(i), arrcity(i), dep(i), arr(i))$$

In the above schedule, *depcity* and *arrcity* are the cities that train i leaves from and arrives in, respectively, and *dep* and *arr* represent the time that it does those two things. Basically, train i leaves $depcity(i)$ at time $dep(i)$, and arrives in $arrcity(i)$ at time $arr(i)$.

Suppose you want to start at time 0 in a particular city, and wanted to see how quickly you could get to each city from your starting city. Write an algorithm to efficiently solve this problem, and explain why it is correct. Remember that this is a train schedule, and so you cannot catch a train if you arrive later at a city than it leaves. The algorithm takes as input the id of the city you start from as well as the 4-tuples.

```
Algorithm:
/* Inputs: "trains" - set of n 4-tuples */
/* First we create the directed weighted graph from the given 4-tuples */
G = {}; /* Directed graph adjacency list */

for train in trains {
    /* Explicitly declaring tuple members for clarity */
    departureCity = train[0];
    arrivalCity = train[1];
    departureTime = train[2];
    arrivalTime = train[3];
    tripDuration = arrivalTime - departureTime;

    /* Add the trip as an outgoing edge to the departure city's node in the adjecency list */
    G[departureCity].append(arrivalCity, tripDuration);
}

/* Now we run a modified version of dijsktra's algorithm $\pi$ = {}; /* hash table, current best dist for v */
d = {}; /* hash table, distance of v */
parents = {}; /* hash table, parents in shortest paths tree */
for v in G do
    $\pi$[v] = $\infty$;
$\pi$[s] = 0, parents[s] = None;
for i = 1 to n do
    u = unfinished node with min $\pi$[u];
    d[u] = $\pi$[u] /* fix distance of u */
    for v in G[u] do
        /* First we make sure that the path is valid due to train arrival/departure times */
        if (G[u][v].departureTime < G[u].arrivalTime)
            continue; // skip the current node
        /* Take into consideration wait time between trains, this will also take into consideration
the wait time for the very first train */
        transferWaitTime = G[u].departureTime - G[parents[u]][u].arrivalTime;
        /* update the distance of neighbors of u */
        if $\pi$[v] > d[u] + G[u][v] + transferWaitTime && transferWaitTime >= 0 then
            $\pi$[v] = d[u] + G[u][v] + transferWaitTime;
            parents[v] = u;
return d, parents
```

Run-time Analysis:
The setup for this algorithm prior to calling the Dijkstras algorithm from lecture will run in time $O(m)$ as it will cycle through every edge in the directed graph and calculate the weight based on the given time. The Dijkstras algorithm itself will run in time $O(nlog(n)) + O(mlog(n))$ as it loops through every node and every edge per node to calculate the shortest path. However, the $O(mlog(n))$ overpowers the $O(nlog(n))$ therefore we can simpfy the Dijkstras overall runtime to just $O(mlog(n))$. As a result the total runtime for this algorithm is $O(m + mlog(n))$.

Proof of Accuracy:
In order to solve this problem, the algorithm is broken down into two main components. The first part uses a for loop to cycle through each station and create an edge weight for every path in between cities. The algorithm uses the departure and arrival times, given as an input of the problem, to calculate and create the weight of every edge. Once the directed graph is created, the Dijkstras algorithm from lecture is implemented on the graph. However, in order to take into account the waiting period in between the arrival and departure at a station, the Dijkstras algorithm was modified to change the weights of each edge by adding the waiting time to the original weight. This includes the time difference between the starting time and the first departure time. In doing so, calling the Dijkstras algorithm, it will find the shortest time taking into account the actual length in between stations as well as the waiting period between trains. Additionally, there is a checker that confirms that the next departure time is after the last arrival time. Hence confirming the path is a valid path and is the shortest possible path from the city initialized as the departure city and the city initialized as the arrival city.